

An extensible formal semantics for UML activity diagrams^{*}

Zamira Daw and Rance Cleaveland

Department of Computer Science, University of Maryland

Abstract. This paper presents an operational semantics for UML activity diagrams. The purpose of this semantics is three-fold: to give a robust basis for verifying model correctness; to help validate model transformations; and to provide a well-formed basis for assessing whether a proposed extension/interpretation of the modeling language is consistent with the standard. The challenges of a general formal framework for UML models include the semi-formality of the semantics specification, the extensibility of the language, and (sometimes deliberate, sometimes accidental) under-specification of model behavior in the standard. Our approach is based on structural operational semantics, which can be extended according to domain-specific needs. The presented semantics has been implemented and tested.

1 Introduction

Model-driven development (MDD) emphasizes the use of models and model transformations through the system development process. Automatic model transformations may be used to generate artifacts for implementation (e.g. code) or for analysis purposes (e.g. intermediate graph representations). This has increased the importance of verification methods for models and model transformations in order to ensure a correct development process. The Unified Modeling Language (UML) [16] has attracted substantial attention as a language for MDD. UML is a non-proprietary, independently maintained standard that includes several graphical sublanguages and a precise abstract syntax given via a metamodel. The semantics is given informally in a natural language, although for some UML sublanguages reference is made to more mathematical models such as state machines and Petri nets. UML also provides an extension mechanism that allows its adaptation to specific application domains.

This work focuses on *UML activity diagrams*, which are generally used to specify the workflow of a system. Activity diagrams can be seen as so-called block diagrams, with a system represented in terms of *actions* (blocks) that compute outputs in terms of inputs, and edges and special-purpose nodes that together determine how data is routed from one action to another. The activity semantics is based on Petri nets semantics. Activity diagrams are among the

^{*} Research supported by NSF Grant CCF-0926194. This information is subject to US-Export Controlled - ECCN:EAR-99

most widely used behavioral diagrams in the UML standard, and find application in a variety of domains, including business-processes engineering [3, 21, 9, 12], embedded systems [10, 6], and medical devices [7].

The purpose of this paper¹ is to present a mathematically well-defined operational semantics for UML activity diagrams. The research is motivated by several concerns. On one hand, to realize the full benefit of MDD, engineers need mechanisms for checking the correctness of their models. Formalizing these checks (e.g. by using model checking) requires a mathematical account of the behavior of diagrams. Reasoning about the correctness of model transformations (e.g. code generation) also requires a precise account of model behavior in order to determine if the transformation correctly preserves model semantics. At the same time, by design, UML may be interpreted flexibly [2, 13] and may be extended via profiles. Determining when an interpretation/extension of UML is semantically consistent with the standard is challenging in the absence of a reference semantics. The challenges are primarily attributable to the following characteristics of the standard: ambiguities, under-specifications, semantic variation points, and semantic extensions using profiling. Figure 1 categorizes the characteristics and gives examples related to activity diagrams. It should be noted that some of the resulting semantic choices are explicitly identified in the standard; others are due to incomplete and sometimes contradictory exposition.

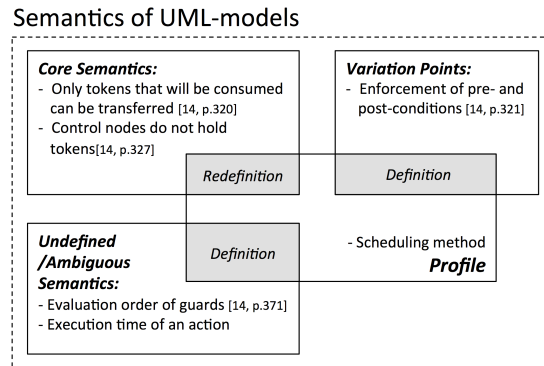


Fig. 1. Semantic components of an interpretation of UML models. Note that profile semantics may redefine the behavior of core concepts, make choices among semantic variation points, and resolve under-specification, as well as introduce new semantic concepts (e.g. scheduling, priority).

Our proposed reference semantics is a structural operational semantics (SOS) (Section 4). SOS is chosen due to its implicit definition translation from models into Kripke structures, which are used in model-verification tools such as model checkers. The proposed semantics uses non-determinism to capture all possible behaviors in case of under-specification or ambiguity in the standard. For ex-

¹ The work is part of a larger effort to formalize UML model semantics in a way that supports the extensibility and flexibility of the standard.

ample, the standard does not define the duration of the execution of an action as is shown in Figure 1. In our semantics, the termination of an action is defined by an inference rule that is applicable after the invocation of the action but is non-deterministically executed with respect to the other applicable rules. Allowing all possible behaviors in this fashion ensures that the proposed semantics covers interpretations consistent with the standard, and that all possible behaviors can be taking into account during the verification. The semantics has been implemented using java language and tested using common activity model structures. Alternative domain-specific semantics can be defined by using the proposed rules, either explicitly or in a customized way, or by adding additional rules (Section 5).

2 Related Work

The work in this paper was initially inspired by research conducted by the authors and another collaborator on analyzing the performance of different model checkers on activity diagrams for medical-device systems [6, 7]. That work required the implementation of translators from diagrams into the (formally precise) input notations of these tools. Other researchers have also confronted this translation problem; examples include [3, 10, 22, 21, 12, 9]. Each of these pieces of work indirectly defines a formal semantics for activities via the translation strategies used. An immediate question that emerges is this: in what sense are these semantics correct, or consistent, with respect to the standard? Providing a means for answering this question is a prime motivation for this paper.

For similar reasons, other researchers have also developed mathematical semantics for (fragments of) UML activity diagrams. A formalization of the earlier UML 1.x standard has been undertaken using Abstract State Machines (ASMs) as a basis in [1]. The more current 2.x standard has been given a semantics in terms of Petri Nets (PN) in [18, 17]. Both approaches may be seen as translational in the sense that the accounts describe how activity diagrams may be interpreted as ASMs / PNs. Another translational approach for UML 2.x is given in [15]; that work interprets activity diagrams by giving a method for converting them into UML state machines. It also gives another account of these diagrams in the action-semantics style of Mosses and Watt. None of these approaches addresses the issue of extensibility nor discusses how to assess an extended semantics against a reference once. They all omit certain aspects of activity-diagram behavior. Among these approaches, the PN-based semantics of Störkle arguably has the broadest coverage of the standard by addressing data flow [19], structured nodes [18], streaming and exceptions [20]. This semantics however does omit some aspects of the standard, such as: token-holding by all nodes, token-transfer limitations, and different types of invocations of activities, which are addressed in our semantics. Störkle also points out some challenges of using a PN-based semantics, such as, the modeling of non-local behaviors (e.g. where tokens of an entire activity or a region have to be terminated, which is presented in final nodes or exceptions), which we believe are easier to address using SOS. In addition, model checkers based on Kripke structures have better capabilities

than PN model checkers. It should be noted that one benefit of a PN semantics is the capability to model so-called *true concurrency*, in which different model behaviors can happen simultaneously. Such features can be captured in SOS also, although we do not do so in this paper for reasons of brevity; instead, our semantics reduces concurrency to interleaving.

Groenniger [11] and Knieke [14] present a flexible semantics for a basic subset of UML activities. Groenniger [11] proposes a denotational inner semantics with variation points. The inner semantics specifies the execution order of actions by using invariant definitions. The type of implementation of actions is defined using variation points (e.g. actions as methods). Our approach allows customizing the implementation of both actions and execution order. Furthermore, the customization range of Groenniger’s approach is limited to its target-domain, which is an object-oriented modeling language. Knieke [14] presents a framework, which enables composition of operational semantics out of fundamental semantic constructs. These constructs define the state and the execution sequence of an activity diagram. The sequence is specified by a step algorithm, which is triggered by a global clock. This algorithm manages the activation of steps that consume or offer tokens. The steps can be customized according to the domain-specific needs. In contrast to our approach, Knieke uses a global clock and a step algorithm that synchronize execution nodes, which is not defined in the standard. Furthermore, the synchronization reduces the number of possible behaviors that can be specified, thereby limiting the set of target-domains. In addition, our approach enables the consistency verification of the extension with the UML standard.

3 Structure of UML activities

The UML activity diagram is a graphical representation of control and data flow. An activity contains nodes and edges. A node represents a function that takes a set of inputs and converts them into a set of outputs. An edge models the connection between two nodes. The execution of a node is determined by its connections and its constraints.

Table 1. Definition of the structure of activity elements

Element type	Definition
Activity	$\langle N, E, APN, PS \rangle$
Action	$\langle I, O, m_{io} \rangle$
CallBehaviorAction	$\langle I, O, behavior, synchronous \rangle$
Fork	$\langle i, O \rangle$
Edge	$\langle source, target, guard, weight \rangle$
Pin	$\langle direction, \delta, upperbound, upper, lower, ordering \rangle$
ActivityParameterNode	$\langle direction, \delta, upperbound, upper, lower, ordering, streaming, exception \rangle$

The structure of activities is defined in the standard by a metamodel, which specifies type of elements, type of connections between elements and elements’ properties that a diagram can have. Table 1 summarizes the properties associated to activity elements that are used in this paper.

An **activity** is composed by a set of nodes N , a set of edges E , and a set of activity parameter nodes APN , which can be grouped in parameter sets PS ($PS \subseteq 2^{APN}$). APN represent the interface of an activity. *Action*, *CallBehaviorAction* and *Fork* are node types. An **action** is the fundamental unit of executable functionality of an activity. An *action* represents a functionality that maps a set of inputs I into a set of outputs O . This mapping is defined outside of the model and represented as $m_{io} : n \rightarrow ((inpin(n) \rightarrow D^*) \rightarrow (outpin(n) \rightarrow D^*))$. A **CallBehaviorAction** invokes a behavior defined by the activity *behavior*. Using this type of node, a system can be built using multiple hierarchical levels. The invocation can be *synchronous* forcing the action to wait until the termination of the behavior's execution. Otherwise, the execution of the action terminates with the invocation of the behavior. A **Fork** is a control node that splits a flow into multiple concurrent flows. This node has one input i and a set of outputs O . **Pins** and **ActivityParameterNodes** (APN) are elements that hold tokens of data type δ and represent inputs and outputs of *actions* or an *activities*, respectively. These elements can be connected by using **edges**². Requirements of the tokens transfer are defined by the *edge* and by the target token holder (*Pin* or APN). On one hand, an *edge* allows transferring tokens that satisfy its *guard* and only if there is at least a minimum amount of tokens (*weight*). On the other hand, a token holder has a limited space (*upperbound*), and accepts only a specific amount of tokens per execution of the corresponding node. This amount is between the *lower* value and the *upper* value. *ordering* defines the order in which tokens have to be saved in the token holder with regard to the arrival (e.g. FIFO). *streaming* allows APN to receive tokens during the execution of the activity. Using *exception* outputs, an activity can yield an exception during the execution.

4 Reference Semantics

This section introduces a reference semantics of activities based on the UML-standard. This paper focuses on the semantics related to tokens transfer, hierarchy, streaming, parameter sets and control nodes. The reference semantics is specified using SOS. This type of semantics describes changes in the system behavior by using labeled transitions ($s \xrightarrow{l} s'$). In the SOS for activities, changes in the state of the model are called steps. Due to the complexity of the activity semantics, we propose two types of steps *micro-steps* and *macro-steps*. While the *macro-steps* involve changes on the state of nodes, *micro-steps* are used to model intermediates transitions.

4.1 State of activity execution

The behavior of an activity is mainly based on coordinated executions of nodes, which is determined by the location of the tokens and the occurrence of events.

² Incoming and outgoing edges are treated as object flows connected between pins of type *ControlToken* (CT) in order to facilitate the semantics specification.

Therefore, the state of a UML model is defined by the status of nodes (S_n), activities (S_a), token holders (S_{th}), and event occurrences (S_Σ). This information is enough to determine all possible following behaviors of the activity in any step of the execution.

Definition 1. Given the model $M = \langle A, \Sigma, D, P_\Sigma \rangle$, where A is a set of activities, Σ is a set of the events names, D is a set of data types and P_Σ is a set of pools that contains occurred events. The state of the system is given by the tuple $\langle S_n, S_a, S_{th}, S_\Sigma \rangle$, where:

- $S_n \in (N \rightarrow \{\text{idle}\} \cup \{\langle \text{executing}, f_{in} \rangle | f_{in} \text{ is a function} \})$, where $\forall n \in N$, if $S_n(n) = \langle \text{executing}, f_{in} \rangle$ then $f_{in} \in (\text{input}(n) \rightarrow D^*)$.
- $S_a \in (A \rightarrow \{\text{idle}\} \cup \{\langle \text{executing}, P_s, P_n \rangle\} \cup \{\langle \text{exception}(v) | v \in D \rangle\})$, where P_s is the parameter set that has invoked the activity, and P_n is a set of APN that have to be set before the activity finishes.
- $S_{th} \in ((P \cup APN) \rightarrow D^*)$, where $\forall h \in (P \cup APN)$, if $S_{th}(h) = V$ then $\forall v \in V. v \in \delta(h)$ and $|V| \leq \text{upperbound}(h)$.
- $S_\Sigma \in (\text{pool} \rightarrow O)$, where O is the set of occurred events in the pool.

4.2 Step semantics

SOS specifies the behavior of a system in terms of inference rules, which determine the valid transitions of the state of the system. An inference rule is applicable if its premises (above a horizontal line) are true and leads to a conclusion (below a horizontal line) that updates the state of the system. The following defines the two proposed steps types.

Definition 2. A macro-step (\rightarrow) is a transition in the state of an activity that leads to a change in the status of a node.

Definition 3. A micro-step (\rightsquigarrow) is a transition in the state of an activity that does not lead to a change in the status of any node.

This distinction is also made in order to facilitate model checking by reducing the state-space, since requirements to verify primarily refer to nodes execution (e.g. $A \square$ (received event $\Rightarrow A \diamond$ motor starts)). Thus, the state-space can be reduced by taking only information about the execution into account. Figure 2(b) shows the state-space on the execution of the actions of figure 2(a). Note that between the end of the execution of action A ($t(A)$) and the beginning of the execution of action B ($i(B)$) or action C ($i(C)$) there are several intermediate states, which represent tokens transfer (e.g. $r(A-B)$). These intermediate states are removed in the reduced state-space (Figure 2(c)).

The reduced state-space is used to verify UML models and the conformity of an interpretation with the standard. In order to differentiate transitions of both state-spaces, transitions in the reduced state-space are called *transitions*. A *transition* is defined as a sequence of *micro-steps* (which can be empty) that ends with a *macro-step* i.e., a transition ends with the start or the finalization of the execution of a node as is shown in definition 4. The *transition* inherits the label l of the *macro-step*.

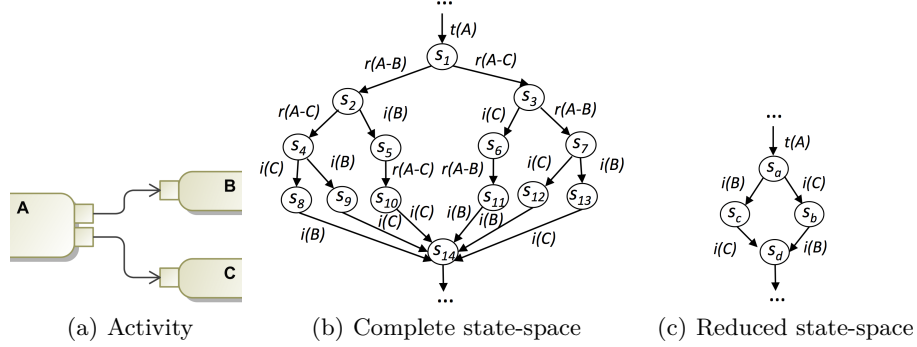


Fig. 2. Reduction of the state-space of an activity by abstracting only the information about nodes execution. Transition labels indicate termination of the execution (t), starting of the execution (i), and transfer of tokens (r)

Transitions have additional requirements (Lines 2-4) related to a subset of control nodes. In the presented work, this sub set is called switch nodes (SN) and is composed of the node types: *Join*, *Fork*, *Merge*, and *Decision*. Since SN cannot hold any token [16, p. 327], the token flow through any SN has to end in the execution of a node not belonging to this subset. This condition is evaluated at the target state of the *macro-step*. N is a set of all nodes in the model.

Definition 4. A transition (\rightarrow) is defined as follow:

$$\begin{array}{rcl}
 \langle S_n, S_a, S_{th}, S_\Sigma \rangle (\sim)^* \langle S'_n, S'_a, S'_{th}, S'_\Sigma \rangle, \langle S'_n, S'_a, S'_{th}, S'_\Sigma \rangle \xrightarrow{l} \langle S''_n, S''_a, S''_{th}, S''_\Sigma \rangle, & 1 \\
 \forall x \in N, type(x) \in SN, \forall p \in inpin(x), S'_{th}(p) = \emptyset, & 2 \\
 \forall y \in N, type(y) \in SN - \{Fork\}, \forall q \in outpin(y), S''_{th}(q) = \emptyset, & 3 \\
 \forall z \in N, type(z) = Fork, \exists p \in outpin(z), S'_{th}(p) = \emptyset & 4 \\
 \hline
 \langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{l} \langle S''_n, S''_a, S''_{th}, S''_\Sigma \rangle & 5
 \end{array}$$

A sequence of SN represents a challenge to the formal specification because it has to be first analyzed whether at the end of the token flow at least one non-SN can be executed and post conditions of the SN are satisfied (Lines 2-4). Therefore, the execution of SN is defined by a *micro-step* in order to analyze all possible token flows of a sequence of SN without changing the state of the activity. This is possible because a *transition* can only end with a *macro-step* and, therefore, no *transition* is created for token flows that do not end in an execution of a non-SN and or do not satisfy constraints for SN.

5 Semantics of model elements

This section presents a set of 37 inference rules that specify the behavior of model element of the activity diagrams defined in the UML standard. This rules have been implemented and tested using a simulator [5].

5.1 Token transfer

Tokens are transfer from a source token holder to a target token holder that are connected by an *edge*. The function *transfer* defines a sequence of tokens, if any, to transfer according to the preconditions, explained in section 3. This function ensures that the size of the sequence is bigger enough to cross the edge (*weight*) and to invoke the target node (*lower*), that the size of the target pin (*upperbound*) is not violated, all transfer tokens are immediately consumed (*upper*), and all tokens of the resulting sequence satisfy the guard.

Definition 5. $transfer : E \rightarrow D^*$

$$transfer(e) = \begin{cases} \emptyset & \text{if } |V_o| < lower(target(e)) \wedge \\ & |V_o| < weight(e) \\ \{v1, \dots, vk\} = \{v \in V_o | k = max\} & \text{elseif } type(node(target(e))) \notin SN \\ \{v1, \dots, vk\} = \{v \in V_o | k \leq max \wedge \\ k \geq lower(target(e)) \wedge \\ k \geq weight(e)\} & \text{else} \end{cases}$$

where, $V_o = \{v \in S_{th}(source(e)) | guard(e)(v) = true\}$, and
 $max = \min(|V_o|, upper(target(t)), upperbound(target(t)))$

SN act as a switch in the token transfer. Therefore, the properties of the transfer are defined by the sources and the targets of the switch, which are the ones that can hold tokens. In order to analyze all possible tokens transfer, the function *transfer* non-deterministically chooses a sequence size (*k*) for SN. The allowed sizes are limited by the preconditions of the token transfer as is shown in the else-statement. In addition to the above-presented preconditions, tokens can be transfer only if they can be immediately consumed by the target node [16, p. 320]. This implies that the consumption of all required tokens and the beginning of the node execution are performed in the same *macro-step*.

5.2 Action

Token consumption and invocation: An example of a *macro-step* is shown in the following SOS-rule, which determines the invocation of an *action*. This rule is applicable only for inactive *actions* that are contained by *activities* that are executing (Line 1). Furthermore, all input pins must have been offered enough tokens for the execution. Note that the function *transfer* determines if the tokens transfer is possible according to the preconditions of the edge, and the target pin. Since an input pin can have multiple *edges*, and thereby multiple source pins, this rule non-deterministically chooses a source by using the function F_{tl} . This function returns a set of injective functions that map target pins into source pins, which offer tokens to consume.

V_{c_i} defines the tokens to consume (Line 3) and V_{q_i} defines the tokens that are in the source pins (Line 6). Thus, the source pins are updated with the

difference (\setminus) of these two sequences (Line 8). The values of the tokens to consume are saved in the action's state using the function f_{in} (Line 4), which maps input pins to sequences of values. This function forms part of the status of the action after the update (Line 7). This function in combination with m_{io} is used in the termination of the action to define which sequence of tokens are offered in the output pins. Note that before the mapping the tokens sequence is reordered (Line 4). After executing this rule, the termination rule of the action becomes applicable.

$$\begin{array}{ll}
a \in A, n \in node(a), type(n) = Action, S_a(a) = \langle executing, P_s, P_n \rangle, S_n(n) = idle, & 1 \\
f_{il} \in F_{il}(inpin(n)), \forall p \in inpin(n). transfer(f_{il}(p)) \neq \emptyset, & 2 \\
\{Vc_1, \dots, Vc_i\} = \{transfer(f_{il}(p)) | \forall p \in inpin(n)\}, & 3 \\
\{p_1, \dots, p_i\} = inpin(n), f_{in} \triangleq ordering(p_k)(Vc_k), \forall 1 \leq k \leq i, & 4 \\
\{q_1, \dots, q_i\} = \{source(f_{il}(p_k)) | \forall 1 \leq k \leq i\}, & 5 \\
\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k) & 6 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle executing, f_{in} \rangle], S_a, & 7 \\
S_{th}[q_1 \mapsto V_{q_1} \setminus Vc_1] \dots [q_i \mapsto V_{q_i} \setminus Vc_i], S_\Sigma \rangle & 8
\end{array}$$

Execution Termination and token offering: This rule is applicable once the action is been executing (Line 2). The outputs are calculated by using the mapping function m_{io} and inputs values (Line 3), which are associated to the state *executing* via the function f_{in} . Note that the resulting tokens have to satisfy the multiplicity of output pins (Lines 7-8).

$$\begin{array}{ll}
a \in A, n \in node(a), type(n) = Action, S_a(a) = \langle executing, P_s, P_n \rangle, & 1 \\
S_n(n) = \langle executing, f_{in} \rangle, & 2 \\
m_{io}(n)(f_{in}) = r \in outpin(n) \rightarrow D^*, & 3 \\
\{q_1, \dots, q_i\} = outpin(n), & 4 \\
\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k), & 5 \\
\forall 1 \leq k \leq i. Vo_k = ordering(q_k)(r(q_k)), & 6 \\
\forall q_k \in outpin(n). |V_{q_k} \cup Vo_k| \leq upperbound(q_k), & 7 \\
\forall q_k \in outpin(n). |Vo_k| \geq lower(q_k) & 8 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto idle], S_a, S_{th}[q_1 \mapsto V_{q_1} \cup Vo_1] \dots [q_i \mapsto V_{q_i} \cup Vo_i], S_\Sigma \rangle & 9
\end{array}$$

5.3 InitialNode

Initial node is invoked during the invocation of the activity to which the node belongs. Therefore, the initial node has only one inference rule.

Termination and offering tokens: Once the node is executing, the termination rule becomes applicable. Initial nodes offers a *ControlToken* in its output after finalizing the execution.

$a \in A, n \in \text{node}(a), \text{type}(n) = \text{InitialNode}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle,$	1
$S_n(n) = \text{executing},$	2
$q = \text{outpin}(n)$	3
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle \text{idle}, \emptyset \rangle], S_a, S_{th}[q \mapsto \{CT\}], S_\Sigma \rangle$	4

5.4 ForkNode

The following SOS-rule specifies token consumption and activation of a *Fork*, which is shown as an example of SN. A *Fork* creates a token in each output for each incoming token. An incoming token is consumed by a *Fork* only if at least one of the outputs offers is accepted, i.e. immediately consumed by the target node. Outgoing tokens that cannot immediately be consumed remain in the output except for tokens that do not satisfy the guard of the outgoing edge. This constraint is also evaluated in the definition of a transition (Definition 4, Line 4). The function *transfer* defines the tokens sequence *Vo* that can be offered by the target node (Line 3). Therefore, this rule is applicable for all possible sizes of sequences returned by this function. *Vc* defines a sub set of the offered tokens that can satisfy the guards of at least one outgoing edge (Line 4). This subset ensures that at least one of the outputs offers is accepted. The precondition of the edge is reevaluated with the set of tokens to consume (Line 5). In case that all premises are satisfied, the state of the pins is updated (Line 11-12). A termination rule sets the sequence of tokens in the outputs (see Appendix).

$a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Fork}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \text{idle},$	1
$e \in E.e = \langle s, t, g, w \rangle \wedge t \in \text{inpin}(n) \wedge \text{transfer}(e) \neq \emptyset,$	2
$Vo = \text{transfer}(e),$	3
$Vc = \{v \in Vo \mid \exists p \in \text{outpin}(n). \exists e \in \text{edge}(a). e = \langle p, t', g', w' \rangle \wedge g'(v) = \text{true}\},$	4
$ Vc \geq \text{weight}(e),$	5
$V_s = S_{th}(\text{source}(e)),$	6
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightsquigarrow \langle S_n[\langle \text{executing}, \emptyset \rangle], S_a, S_{th}[\text{source}(e) \mapsto V_s] Vc]$	7
$[\text{target}(e) \mapsto Vc], S_\Sigma \rangle$	8

Termination and token offering: Tokens that satisfy the guards are offered in the outputs (Line 6). Note that invocation rule of the *Fork* (Section ??) ensures that all tokens satisfy at least the guard of one output.

$a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Fork}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle,$	1
$S_n(n) = \langle \text{executing}, \emptyset \rangle,$	2
$Vo = S_{th}(\text{inpin}(n)),$	3
$\{q_1, \dots, q_i\} = \text{outpin}(n),$	4
$\{V_{q_1}, \dots, V_{q_i}\} \subseteq D^*,$	5
$\forall 1 \leq k \leq i. V_{q_k} = \{v \in Vo \mid \exists e \in \text{edge}(a). e = \langle q_k, t', g', w' \rangle \wedge g'(v) = \text{true}\},$	6
$\forall 1 \leq k \leq i. V_{r_k} = S_{th}(q_k),$	7
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightsquigarrow \langle S_n[n \mapsto \text{idle}], S_a, S_{th}[\text{inpin}(n) \mapsto \emptyset]$	8
$[q_1 \mapsto V_{r_1} \cup V_{q_1}] \dots [q_i \mapsto V_{r_i} \cup V_{q_i}], S_\Sigma \rangle$	9

5.5 JoinNode

Token consumption and invocation: A join node is executed if the corresponding *JoinSpecification* is true (Line 4). This specification determines which inputs are required to have tokens for the execution. If multiple *ControlTokens* are offered in one input, the token are merged into one tokens by using the function *combine*. Data tokens that are offered in different inputs are offered in the output as an ordered set. The order is given by the order of offering, which is defined in the state by P .

$$\begin{array}{l}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Join}, \\
S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \langle \text{idle}, P \rangle, \\
\text{joinSpec}(\text{inpin}(n)) = \text{true}, \\
f_{tl} \in F_{tl}(\text{inpin}(n)), \forall p \in \text{inpin}(n). \text{transfer}(f_{tl}(p)) \neq \emptyset, \\
\forall p \in \text{inpin}(n). \delta(p) \neq \text{ControlToken} \wedge p \in P, \\
\{V_{c_{p_1}}, \dots, V_{c_{p_i}}\} = \{\text{combine}(\text{transfer}(f_{tl}(p))) \mid \forall t \in \text{inpin}(p)\}, \\
\{p_1, \dots, p_i\} = \text{inpin}(n), f_{in} \triangleq \text{index}(P, p_k), \forall 1 \leq k \leq i, \\
\{q_1, \dots, q_i\} = \{s \mid \forall t \in \text{inpin}(p). \exists e = \langle s, t, g, w \rangle\}, \\
\forall 1 \leq k \leq i. V_{c_{p_k}} = \text{combine}(\text{transfer}(f_{tl}(p_k))), \\
\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k) \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle \text{executing}, f_{in} \rangle], S_a, S_{th}[p_1 \mapsto V_{c_{p_1}}] \dots [p_i \mapsto V_{c_{p_i}}] \\
[q_1 \mapsto V_{q_1}] \dots [q_i \mapsto V_{q_i}] V_{c_{p_i}}, S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10 \\
11 \\
12 \\
13
\end{array}$$

Saving the order of token offering - adding: This rule aims to save the order in which the tokens are offered by saving the pin in the state of the node.

$$\begin{array}{l}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Join}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
S_n(n) = \langle \text{idle}, P_{\text{order}} \rangle, \\
p \in \text{inpin}(n). p \notin P_{\text{order}} \wedge p \neq \text{ControlToken} \wedge \\
\exists e \in \text{edge}(a). e = \langle s, p, g, w \rangle \wedge \text{transfer}(e) \neq \emptyset \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{\tau} \langle S_n[n \mapsto \langle \text{idle}, P_{\text{order}} \cup p \rangle], S_a, S_{th}, S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5
\end{array}$$

Saving the order of token offering - removing: This rule is applicable in case the a tokens that have been offered to the join node has been consumed by other node. This is necessary in order to update the list of offering pins from the state of the node.

$$\begin{array}{l}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Join}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
S_n(n) = \langle \text{idle}, P_{\text{order}} \rangle, \\
p \in \text{inpin}(n). p \in P_{\text{order}} \wedge p \neq \text{ControlToken} \wedge \\
\exists e \in \text{edge}(a). e = \langle s, p, g, w \rangle \wedge \text{transfer}(e) = \emptyset \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{\tau} \langle S_n[n \mapsto \langle \text{idle}, P_{\text{order}} \setminus p \rangle], S_a, S_{th}, S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5
\end{array}$$

5.6 MergeNode

In contrast with other nodes, merge nodes required only one rule to transfer input tokens to outgoing outputs.

Invocation, and token consumption and offering Note that the transfer is carried out without invoking the node.

$$\begin{array}{ll}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Merge}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \text{idle}, & 1 \\
e \in E, e = \langle s, t, g, w \rangle \wedge t \in \text{inpin}(n) \wedge \text{transfer}(e) \neq \emptyset, & 2 \\
Vo = \text{transfer}(e), & 3 \\
p = \text{source}(e), q \in \text{outpin}(n), & 4 \\
V_q = \{V | S_{th}(q) = V\}, & 5 \\
V_p = \{V | S_{th}(p) = V\}, & 6 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n, S_a, S_{th}[q \mapsto V_q \cup Vo][p \mapsto V_p]Vc], S_\Sigma \rangle & 7
\end{array}$$

5.7 DecisionNode

The entire behavior of a decision node is specified in 7 inference rules. This is because the value that is evaluated in the decision can come from the input of the node, a special input or from the result of an associated behavior.

Token consumption and invocation: This rule invokes the decision node.

$$\begin{array}{ll}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Decision}, & 1 \\
S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \text{idle}, & 2 \\
f_{tl} \in F_{tl}(\text{inpin}(n)), \forall p \in \text{inpin}(n). \text{transfer}(f_{tl}(p)) \neq \emptyset, & 3 \\
\{p_1, \dots, p_i\} = \text{inpin}(n), & 4 \\
\{q_1, \dots, q_i\} = \{\text{source}(f_{tl}(p_k)) | \forall 1 \leq k \leq 2\}, & 5 \\
\forall 1 \leq k \leq i. Vc_k = \text{transfer}(f_{tl}(p_k)), & 6 \\
\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k), & 7 \\
\forall 1 \leq k \leq i. |Vc_1| = |Vc_k| & 8 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle \text{executing}, \emptyset \rangle], S_a, S_{th}[p_1 \mapsto Vc_1] \dots [p_i \mapsto Vc_i] & 9 \\
[q_1 \mapsto V_{q_1}]Vc_1] \dots [q_i \mapsto V_{q_i}]Vc_i], S_\Sigma \rangle & 10
\end{array}$$

Termination: This rule terminates the execution of the decision node. Note that this rule is applicable only when there is not token in its input. This token is consumed after the guard evaluation is executed.

$$\begin{array}{ll}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Decision}, & 1 \\
S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, & 2 \\
S_{th}(\text{inpin}(n)) = \emptyset, S_a(d\text{Behavior}(n)) = \text{idle} & 3 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto \text{idle}], S_a, S_{th}, S_\Sigma \rangle & 4
\end{array}$$

Guard evaluation over the input values and token offering: This micro step removes the token from the input pin and offers it to the edge, whose guards evaluate true for the input value.

$$\begin{array}{l}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Decision}, \\
S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, \\
dFlow = \emptyset, dBehavior = \emptyset, S_{th}(\text{inpin}(n)) \neq \emptyset, \\
V_p = \{v_{p_1}, \dots, v_{p_i}\} = S_{th}(\text{inpin}(n)) = V, \\
r \in \text{outpin}(n). \exists e \in E. e = \langle r, t, g, w \rangle \wedge g(v_{p_1}) = \text{true}, \\
V_r = S_{th}(r) \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{\tau} \langle S_n, S_a, S_{th}[\text{inpin}(n) \mapsto V_p] \{v_{p_1}\} [r \mapsto V_r \cup \{v_{p_1}\}], S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7
\end{array}$$

Guard evaluation over $dFlow$ and token offering: In contrast to previous rule, this rule evaluates the guard over the value of the pin $dFlow$

$$\begin{array}{l}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Decision}, \\
S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, \\
dFlow \neq \emptyset, dBehavior = \emptyset, S_{th}(\text{inpin}(n)) \neq \emptyset, \\
q = dFlow(n), p = \text{inpin}(n) - q, \\
V_p = \{v_{p_1}, \dots, v_{p_i}\} = S_{th}(p), \\
V_q = \{v_{q_1}, \dots, v_{q_i}\} = S_{th}(q), \\
r \in \text{outpin}(n). \exists e \in E. e = \langle r, t, g, w \rangle \wedge g(v_{q_1}) = \text{true}, \\
V_r = S_{th}(r) \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{\tau} \langle S_n, S_a, S_{th}[\text{inpin}(n) \mapsto V_p] \{v_{p_1}\} [r \mapsto V_r \cup \{v_{p_1}\}] \\
[q \mapsto V_q] \{v_{q_1}\}], S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9 \\
10
\end{array}$$

Guard evaluation over the result of $dBehavior$ and token offering: In this rule, the guard is evaluated over the value that returns the behavior $dBehavior$.

$$\begin{array}{l}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Decision}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
S_n(n) = \langle \text{executing}, \emptyset \rangle, dBehavior \neq \emptyset, S_{th}(\text{inpin}(n)) \neq \emptyset, \\
V_p = \{v_{p_1}, \dots, v_{p_i}\} = S_{th}(\text{outpin}(dBehavior)) = V, \\
r \in \text{outpin}(n). \exists e \in E. e = \langle r, t, g, w \rangle \wedge g(v_{p_1}) = \text{true}, \\
V_r = S_{th}(r) \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{\tau} \langle S_n, S_a, S_{th}[\text{inpin}(n) \mapsto V_p] \{v_{p_1}\} [r \mapsto V_r \cup \{v_{p_1}\}], S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5 \\
6
\end{array}$$

Invoking decision behavior: In case that the decision node has a *dBehavior*, the behavior is invoked by tranfering the input value of the decision node to the behavior.

$$\begin{array}{ll}
a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{Decision}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, & 1 \\
S_n(n) = \langle \text{executing}, \emptyset \rangle, d\text{Behavior} \neq \emptyset, & 2 \\
S_a(d\text{Behavior}(n)) = \text{idle}, S_{th}(\text{inpin}(n)) \neq \emptyset, S_{th}(\text{outpin}(d\text{Behavior})) = \emptyset & 3 \\
\forall p \in \text{inpin}(d\text{Behavior}(n)).S_{th}(p) = \emptyset, & 4 \\
\{p_1, \dots, p_i\} = \text{inpin}(n), & 5 \\
\{V_{p_1}, \dots, V_{p_i}\} = V_p = \{V | S_{th}(p) = V\}, & 6 \\
\forall 1 \leq k \leq i. \{v_{p_{k_1}}, \dots, v_{p_{k_j}}\} = V_{p_k} & 7 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{\tau} \langle S_n, S_a, S_{th}[m_p(p_1) \mapsto \{v_{p_{1_1}}\}][m_p(p_i) \mapsto \{v_{p_{i_1}}\}], S_\Sigma \rangle & 8
\end{array}$$

Termination and token offering for dBehavior : After the termination of the *dBehavior*, the result is tranfered to the pin that is used to evaluate the guard.

$$\begin{array}{ll}
a, b \in A, n \in \text{node}(a), \text{type}(n) = \text{Decision}, b = d\text{Behavior}(n), & 1 \\
S_a(a) = \langle \text{executing}, P'_s, P'_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, S_a(b) = \langle \text{executing}, P_s, \emptyset \rangle, & 2 \\
\forall th \in (\text{holders}(b) - \text{output}(b)).S_{th}(p) = \emptyset, & 3 \\
\forall m \in \text{node}(b).S_n(m) = \text{idle}, & 4 \\
o = \text{output}(b) & 5 \\
V_o = \{V | S_{th}(o) = V\} & 6 \\
r \in \text{outpin}(n). \exists e \in E. e = \langle r, t, g, w \rangle \wedge \forall v \in V_o. g(v) = \text{true}, & 7 \\
V_r = \{V | S_{th}(r) = V\} & 8 \\
\{p_1, \dots, p_i\} = \text{inpin}(n), & 9 \\
\{V_{p_1}, \dots, V_{p_i}\} = V_p = \{V | S_{th}(p) = V\}, & 10 \\
\forall 1 \leq k \leq i. \{v_{p_{k_1}}, \dots, v_{p_{k_j}}\} = V_{p_k} & 11 \\
\{v_{q_1}, \dots, v_{q_i}\} = V_q = \{v \in V | S_{th}(q) = V \wedge q = \text{inpin}(n) - d\text{Flow}(n)\}, & 12 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto \text{idle}], S_a[b \mapsto \text{idle}], S_{th}[m_p(q_1) \mapsto V_{q_1} \cup V_{p_1}] \dots & 13 \\
[m_p(p_i) \mapsto V_{p_i} \cup V_{p_i}][m_p(r_1) \mapsto V_{r_1} \cup \{\text{null}\}] \dots & 14 \\
[m_p(r_j) \mapsto V_{r_j} \cup \{\text{null}\}][p_1 \mapsto \emptyset] \dots [p_i \mapsto \emptyset], S_\Sigma \rangle & 15
\end{array}$$

5.8 FlowFinalNode

This node does not have any outputs and consume all input tokens.

Token consumption and invocation:

$a \in A, n \in \text{node}(a), \text{type}(n) = \text{FlowFinalNode},$	1
$S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \text{idle},$	2
$e \in E.e = \langle s, t, g, w \rangle \wedge t \in \text{inpin}(n) \wedge \text{transfer}(e) \neq \emptyset,$	3
$Vc = \text{transfer}(e),$	4
$V_s = \{V S_{th}(\text{source}(e)) = V\}$	5
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle \text{executing}, \emptyset \rangle], S_a, [\text{source}(e) \mapsto V_s] Vc, S_\Sigma \rangle$	6

Execution termination:

$a \in A, n \in \text{node}(a), \text{type}(n) = \text{FlowFinalNode}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle,$	1
$S_n(n) = \langle \text{executing}, \emptyset \rangle,$	2
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto \text{idle}], S_a, S_{th}, S_\Sigma \rangle$	3

5.9 ActivityFinalNode

This node terminates all processing of the activity in its invocation

Token consumption and invocation for asynchronous call:

$a \in A, n, m \in \text{node}(a), \text{type}(n) = \text{ActivityFinalNode},$	1
$S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \text{idle},$	2
$\text{type}(m) = \text{CallBehaviorAction}, a = \text{behavior}(m), \text{isSynchronous}(m) = \text{false},$	3
$e \in E.e = \langle s, t, g, w \rangle \wedge t \in \text{inpin}(n) \wedge \text{transfer}(e) \neq \emptyset,$	4
$B_{\text{async}} = \{b \in A \exists n \in \text{node}(a) \wedge b = \text{behavior}(n) \wedge \text{isSynchronous}(n) = \text{false}\},$	5
$\{n_1, \dots, n_i\} = \text{node}(a) - \{n \in \text{node}(b) b \in B_{\text{async}}\},$	6
$\{a_1, \dots, a_j\} = a \cup \{b \in A b \neq a \wedge \exists n \in \text{node}(a) \wedge b = \text{behavior}(n) \wedge$	7
$\text{isSynchronous}(n) = \text{true}\},$	8
$\{p_1, \dots, p_k\} = \text{pin}(a) - \{p \in \text{pin}(b) b \in B_{\text{async}}\} \quad q$	9
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n_1 \mapsto \text{idle}] \dots [n_i \mapsto \text{idle}], S_a[a_1 \mapsto \text{idle}] \dots [a_l \mapsto \text{idle}],$	10
$S_{th}[p_1 \mapsto \emptyset] \dots [p_k \mapsto \emptyset], S_\Sigma \rangle$	11

Token consumption and invocation for synchronous call:

$a \in A, n \in node(a), type(n) = ActivityFinalNode,$	1
$S_a(a) = \langle executing, P_s, P_n \rangle, S_n(n) = idle,$	2
$type(m) = CallBehaviorAction, a = behavior(m), isSynchronous(m) = true,$	3
$e \in E.e = \langle s, t, g, w \rangle \wedge t \in inpin(n) \wedge transfer(e) \neq \emptyset,$	4
$B_{async} = \{b \in A \exists n \in node(a) \wedge b = behavior(n) \wedge isSynchronous(n) = false\},$	5
$\{n_1, \dots, n_i\} = node(a) - \{n \in node(b) b \in B_{async}\},$	6
$\{a_1, \dots, a_j\} = a \cup \{b \in A b \neq a \wedge \exists n \in node(a) \wedge b = behavior(n) \wedge$	7
$isSynchronous(n) = true\},$	8
$\{t_1, \dots, t_k\} = pin(a) - \{p \in pin(b) b \in B_{async}\},$	9
$P = \{p \in outpin(a) S_{th} = V \wedge V \neq \emptyset\}, P' = outpin(a) - P,$	10
$\{q_1, \dots, q_i\} = \{p \in inpin(m) m_p^{-1}(p) \in P\},$	11
$\{r_1, \dots, r_j\} = \{p \in inpin(m) m_p^{-1}(p) \notin P\},$	12
$\{V_{p_1}, \dots, V_{p_i}\} = \{V \forall p \in P. S_{th}(p) = V\},$	13
$\{V_{q_1}, \dots, V_{q_i}\} = \{V \forall q \in P. S_{th}(m_p^{-1}(q)) = V\},$	14
$\{V_{r_1}, \dots, V_{r_j}\} = \{V \forall r \in P'. S_{th}(m_p^{-1}(r)) = V\},$	15
$\forall q_k \in m_p^{-1}(P). V_{q_k} \cup V_{p_k} \leq upperbound(q_k),$	16
$\forall r_k \in m_p^{-1}(P'). V_{r_k} + 1 \leq upperbound(r_k),$	17
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n_1 \mapsto idle] \dots [n_i \mapsto idle][m \mapsto idle],$	18
$S_a[a_1 \mapsto idle] \dots [a_l \mapsto idle], S_{th}[t_1 \mapsto \emptyset] \dots [t_k \mapsto \emptyset][q_i \mapsto V_{q_i} \cup V_{p_i}]$	19
$[r_1 \mapsto V_{r_1} \cup \{null\}] \dots [r_j \mapsto V_{r_j} \cup \{null\}], S_\Sigma \rangle$	20

5.10 AcceptEventAction

AcceptEventActions have different behaviors when they have inputs or not, and therefore, the node need more rules to be specified. A node without pins are activated when the activity is invoked.

Token consumption and invocation for a node with inpins: The node is invoked when the required amount of tokens has been offered.

$a \in A, n \in node(a), type(n) = AcceptEventAction,$	1
$S_a(a) = \langle executing, P_s, P_n \rangle, S_n(n) = idle,$	2
$inpin(n) \neq \emptyset,$	3
$f_{tl} \in F_{tl}(inpin(n)), \forall p \in inpin(n). transfer(f_{tl}(p)) \neq \emptyset,$	4
$\{p_1, \dots, p_i\} = inpin(n),$	5
$\{q_1, \dots, q_i\} = \{source(f_{tl}(p_k)) \forall 1 \leq k \leq i\},$	6
$\{V_{c_1}, \dots, V_{c_i}\} = \{transfer(f_{tl}(p)) \forall p \in inpin(n)\},$	7
$\{V_{q_1}, \dots, V_{q_i}\} = \{V \forall 1 \leq k \leq i. S_{th}(q_k) = V\}$	8
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle executing, \emptyset \rangle], S_a,$	9
$S_{th}[q_1 \mapsto V_{q_1} \sqcup V_{c_1}] \dots [q_i \mapsto V_{q_i} \sqcup V_{c_i}], S_\Sigma \rangle$	10

Receiving events, termination and offering tokens: This rule is only applicable for node that have *inpin*. Once the node has received the corresponding event, the execution of the node is terminated by this rule. Thus, the node is not able to receive more events.

$$\begin{array}{ll}
a \in A, n \in \text{node}(a), \text{type}(n) = \text{AcceptEventAction}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, & 1 \\
S_n(n) = \langle \text{executing}, \emptyset \rangle, \text{inpin}(n) \neq \emptyset, & 2 \\
\langle \sigma, v_e \rangle \in V.\text{event}(n) = \sigma, & 3 \\
r = \text{result}(n), \{p_1, \dots, p_i\} = \text{outpin}(n) - r, & 4 \\
\{V_{p_1}, \dots, V_{p_i}\} = \{V \mid \forall 1 \leq k \leq i. S_{th}(p_k) = V\}, & 5 \\
V_r = \{V \mid S_{th}(r) = V\}, & 6 \\
|V_r| + 1 \leq \text{upperbound}(r) \wedge \forall 1 \leq k \leq i. |V_{p_k}| + 1 \leq \text{upperbound}(p_k) & 7 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto \text{idle}], S_a, S_{th}[r \mapsto V_r \cup \{v_e\}] & 8 \\
[p_1 \mapsto V_{p_1} \cup \{\text{ControlToken}\}] \dots [p_i \mapsto V_{p_i} \cup \{\text{ControlToken}\}], S_\Sigma \rangle & 9
\end{array}$$

Receiving events and offering tokens without termination: In contrast to previous rule, the execution of *AcceptEventActions* is not terminated after receiving an event. These node are executing as long as the activity is executing.

$$\begin{array}{ll}
a \in A, n \in \text{node}(a), \text{type}(n) = \text{AcceptEventAction}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, & 1 \\
S_n(n) = \langle \text{executing}, \emptyset \rangle, \text{inpin}(n) = \emptyset, & 2 \\
\langle \sigma, v_e \rangle \in V.\text{event}(n) = \sigma, & 3 \\
r = \text{result}(n), \{p_1, \dots, p_i\} = \text{outpin}(n) - r, & 4 \\
\{V_{p_1}, \dots, V_{p_i}\} = \{V \mid \forall 1 \leq k \leq i. S_{th}(p_k) = V\}, & 5 \\
V_r = \{V \mid S_{th}(r) = V\}, & 6 \\
|V_r| + 1 \leq \text{upperbound}(r) \wedge \forall 1 \leq k \leq i. |V_{p_k}| + 1 \leq \text{upperbound}(p_k) & 7 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n, S_a, S_{th}[r \mapsto V_r \cup \{v_e\}] & 8 \\
[p_1 \mapsto V_{p_1} \cup \{\text{ControlToken}\}] \dots [p_i \mapsto V_{p_i} \cup \{\text{ControlToken}\}], S_\Sigma \rangle & 9
\end{array}$$

5.11 SendSignalAction

Token consumption, invocation, sending of the signal: Once the invocation requirements of the node are satisfied, a event is saved in the event pool.

$a \in A, n \in \text{node}(a), \text{type}(n) = \text{SendSignalAction},$	1
$S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \text{idle},$	2
$f_{tl} \in F_{tl}(\text{inpin}(n)), \forall p \in \text{inpin}(n). \text{transfer}(f_{tl}(p)) \neq \emptyset,$	3
$\{V_{c1}, \dots, V_{ci}\} = \{\text{transfer}(f_{tl}(p)) \forall p \in \text{inpin}(n)\},$	4
$\{q_1, \dots, q_i\} = \{\text{source}(f_{tl}(p_k)) \forall 1 \leq k \leq i\},$	5
$\{V_{q1}, \dots, V_{qi}\} = \{V \forall 1 \leq k \leq i. S_{th}(q_k) = V\}$	6
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle \text{executing}, \emptyset \rangle], S_a, S_{th}[q_1 \mapsto V_{q1} \mid V_{c1}] \dots$	7
$[q_i \mapsto V_{qi} \mid V_{ci}], S_\Sigma \cup \langle \text{event}(n), \text{sendData}(V_{c1}, \dots, V_{ci}) \rangle$	8

Execution Termination and token offering:

$a \in A, n \in \text{node}(a), \text{type}(n) = \text{SendSignalAction},$	1
$S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle,$	2
$\{q_1, \dots, q_i\} = \text{outpin}(n),$	3
$\{V_{q1}, \dots, V_{qi}\} = \{V \forall q \in \text{outpin}(n). S_{th}(q) = V\},$	4
$\forall 1 \leq k. V_{qk} < \text{upperbound}(q_k)$	5
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto \text{idle}], S_a, S_{th}[q_1 \mapsto V_{q1} \cup \text{ControlToken}] \dots$	6
$[q_i \mapsto V_{qi} \cup \text{ControlToken}], S_\Sigma \rangle$	7

5.12 CallBehaviorAction

Multiple hierarchical levels can be built by using *CallBehaviorActions*. This type of node invokes an activity by making tokens available to the activity that are offered to the node. If the node is *synchronous*, the execution of the node terminates only when the corresponding activity execution is finished (see Appendix). The results of the execution of the activity are offered in the outputs of the node. Furthermore, *pins* of the nodes inherit properties of the *APN* of the activity, such as, parameter set, *streaming*, and *exception*.

Token consumption and invocation with non-streaming and streaming parameters:

While tokens can be consumed by streaming parameters during the execution of the activity, tokens offered to non-streaming parameter are required to invoke the activity and are only consuming at the invocation point [16, p. 410]. P_{req} defines the set of pins of the node that are required for invoking the activity (Line 4). m_p maps *pins* of a *CallBehaviorAction* to the *APN* of the activity. Note that all required pins belong to the same *PS* of the activity. If multiple *PS* have enough tokens to start the invocation, one of them is chosen non-deterministically. This rule is applicable only if all required pins have been offered

enough tokens (Line 5). P_{cons} contains streaming and non-streaming pins that have been offered tokens belonging to the chosen PS (Line 6). Tokens offered to these pins are consumed by the invocation (Lines 8,14). The sequence of tokens to consume is defined by V_{c_i} (Line 10). The APN are updated by adding the consumed tokens (Line 14). This rule invokes the node and transfer tokens to the activity. An extra rule is used to invoke the activity.

The execution of an activity starts when any input has a token. Requirements of the activity are evaluated in the invocation of the *CallBehaviorAction*. Tokens flow and events reception is started by invoking *InitialNodes* and *AcceptEventActions*, respectively.

$a, b \in A, n \in node(a), S_a(a) = \langle executing, P_s, P_n \rangle, S_n(n) = idle, S_a(b) = idle,$	1
$type(n) = CallBehaviorAction, isSynchronous(n) = true, b = behavior(n),$	2
$PS_b \in PS(b). \exists th \in PS_b, streaming(th) = FALSE,$	3
$P_{req} = \{p \in inpin(n) th = m_p(p) \wedge th \in PS_b \wedge streaming(th) = FALSE\},$	4
$f_{tl} \in F_{tl}(inpin(n)), \forall p \in P_{req}. transfer(f_{tl}(p)) \neq \emptyset,$	5
$P_{cons} = \{p \in inpin(n) m_p(p) \in PS_b \wedge transfer(f_{tl}(p)) \neq \emptyset\},$	6
$\{p_1, \dots, p_i\} = P_{cons},$	7
$\{q_1, \dots, q_i\} = \{source(f_{tl}(p_k)) \forall 1 \leq k \leq i\},$	8
$\{r_1, \dots, r_i\} = \{m_p(p_k) \forall 1 \leq k \leq i\},$	9
$\{V_{c_1}, \dots, V_{c_i}\} = \{transfer(f_{tl}(p)) \forall p \in P_{cons}\},$	10
$\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k),$	11
$\forall 1 \leq k \leq i. V_{r_k} = S_{th}(r_k)$	12
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle executing, \emptyset \rangle], S_a,$	13
$S_{th}[q_1 \mapsto V_{q_1} \upharpoonright V_{c_1}] \dots [q_i \mapsto V_{q_i} \upharpoonright V_{c_i}][r_1 \mapsto V_{r_1} \cup V_{c_1}] \dots [r_i \mapsto V_{r_i} \cup V_{c_i}], S_\Sigma \rangle$	14

Token consumption and invocation with only streaming parameter:

If there is only streaming parameter, the *Activity* is invoked if at least one input has been offered a token.

$a, b \in A, n \in node(a), type(n) = CallBehaviorAction, isSynchronous(n) = true,$	1
$S_a(a) = \langle executing, P_s, P_n \rangle, S_n(n) = idle, b = behavior(n), S_a(b) = idle,$	2
$PS_b \in PS(b). \forall th \in PS_b, streaming(th) = TRUE,$	3
$f_{tl} \in F_{tl}(inpin(n)). \exists p \in inpin(n). m_p(p) \in PS_b \wedge transfer(f_{tl}(p)) \neq \emptyset,$	4
$P_{cons} = \{p \in inpin(n) m_p(p) \in PS_b \wedge transfer(f_{tl}(p)) \neq \emptyset\},$	5
$\{p_1, \dots, p_i\} = P_{cons},$	6
$\{q_1, \dots, q_i\} = \{th \in PS_b m_p^{-1}(th) \in P_{cons}\},$	7
$\{V_{q_1}, \dots, V_{q_i}\} = \{V_k \forall 1 \leq k \leq i. S_{th}(q_k) = V_k\},$	8
$\{V_{r_1}, \dots, V_{r_i}\} = \{V \forall p \in P_{cons}. S_{th}(f_{tl}(p)) = V\},$	9
$\{V_{c_1}, \dots, V_{c_i}\} = \{transfer(f_{tl}(p)) \forall p \in P_{cons}\},$	10
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[n \mapsto \langle executing, \emptyset \rangle], S_a, S_{th}[f_{tl}(p_1) \mapsto V_{r_1} \upharpoonright V_{c_1}] \dots$	11
$[f_{tl}(p_i) \mapsto V_{r_i} \upharpoonright V_{c_i}], [q_1 \mapsto V_{q_1} \upharpoonright V_{c_1}] \dots [q_i \mapsto V_{q_i} \upharpoonright V_{c_i}], S_\Sigma \rangle$	12

Token consumption during the execution from streaming parameters: Tokens are passed from the *CallBehaviorAction* to the *Activity* during the execution if they have been offered in a input streaming pin.

$$\begin{array}{ll}
a, b \in A, n \in \text{node}(a), \text{type}(n) = \text{CallBehaviorAction}, b = \text{behavior}(n), & 1 \\
S_a(a) = \langle \text{executing}, P'_s, P'_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, S_a(b) = \langle \text{executing}, P_s, P_n \rangle, & 2 \\
\text{isSynchronous}(n) = \text{true}, & 3 \\
P_{\text{stream}} = \{p \in \text{inpin}(n) \mid \text{th} = m_p(p) \wedge \text{th} \in P_s \wedge \text{streaming}(\text{th}) = \text{true}\}, & 4 \\
f_{tl} \in F_{tl}(\text{inpin}(p)). \exists p \in P_{\text{stream}}. \text{transfer}(f_{tl}(p)) \neq \emptyset, & 5 \\
P_{\text{cons}} = \{p \in P_{\text{stream}} \mid \text{transfer}(f_{tl}(p)) \neq \emptyset\}, & 6 \\
\{p_1, \dots, p_i\} = P_{\text{cons}}, & 7 \\
\{q_1, \dots, q_i\} = \{\text{th} \in P_s \mid m_p^{-1}(\text{th}) \in P_{\text{cons}}\}, & 8 \\
\{r_1, \dots, r_i\} = \{\text{source}(f_{tl}(p_k)) \mid \forall 1 \leq k \leq i\}, & 9 \\
\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k), & 10 \\
\forall 1 \leq k \leq i. V_{r_k} = S_{th}(r_k), & 11 \\
\forall 1 \leq k \leq i. V_{c_k} = S_{th}(\text{transfer}(f_{tl}(p_k))) & 12 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n, S_a[b \mapsto \langle \text{executing}, P_s, P_n - P_{\text{cons}} \rangle], & 13 \\
S_{th}[r_1 \mapsto V_{r_1} \mid V_{c_1}] \dots [r_i \mapsto V_{r_i} \mid V_{c_i}][q_1 \mapsto V_{q_1} \cup V_{c_1}] \dots [q_i \mapsto V_{q_i} \cup V_{c_i}], S_\Sigma \rangle & 14
\end{array}$$

Token offering of streaming outputs: Tokens in output streaming parameters are tranfered to the *CallBehaviorAction* during the execution.

$$\begin{array}{ll}
a, b \in A, n \in \text{node}(a), \text{type}(n) = \text{CallBehaviorAction}, b = \text{behavior}(n), & 1 \\
S_a(a) = \langle \text{executing}, P'_s, P'_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, S_a(b) = \langle \text{executing}, P_s, P_n \rangle, & 2 \\
\text{isSynchronous}(n) = \text{true}, & 3 \\
p \in \text{output}(b). S_{th} \neq \emptyset \wedge \text{streaming}(\text{th}) = \text{true}, & 4 \\
q = m_p^{-1}(p), & 5 \\
V_p = S_{th}(p), & 6 \\
V_q = S_{th}(q), & 7 \\
|V_q \cup V_p| \leq \text{upperbound}(q) & 8 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n, S_a, S_{th}[p \mapsto \emptyset][q \mapsto V_q \cup V_p], S_\Sigma \rangle & 9
\end{array}$$

5.13 Activity

Termination and token offering of an activity: This rule is applicable when there are no tokens left to process in the activity or nodes that are been executing (Lines 4,5). Since this rule is for a synchronous call (Line 3), the resulting tokens are offered in the outpins of the corresponding *CallBehaviorAction*, which is also terminated in this rule (Line 15). Note that a *null* token is offered in case that the corresponding outpin is empty by the end of the activity (Lines 9,17).

$a, b \in A, n \in \text{node}(a), \text{type}(n) = \text{CallBehaviorAction}, b = \text{behavior}(n),$	1
$S_a(a) = \langle \text{executing}, P'_s, P'_n \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, S_a(b) = \langle \text{executing}, P_s, \emptyset \rangle,$	2
$\text{isSynchronous}(n) = \text{true},$	3
$\forall th \in (\text{holders}(b) - \text{output}(b)).S_{th}(p) = \emptyset,$	4
$\forall m \in \text{node}(b).S_n(m) = \text{idle},$	5
$P_{ocu} = \{th \in \text{output}(b) S_{th}(th) \neq \emptyset \wedge \text{isExeption}(th) = \text{false}\},$	6
$P_s \in PS(b). \forall p \in P_{ocu}. p \in P_s$	7
$\{p_1, \dots, p_i\} = P = \{p \in P_s S_{th} \neq \emptyset\},$	8
$\{r_1, \dots, r_j\} = P' = P_s - P,$	9
$\forall 1 \leq k \leq i. V_{p_k} = S_{th}(p_k),$	10
$\forall 1 \leq k \leq i. V_{q_k} = S_{th}(m_p(q_k)),$	11
$\forall 1 \leq k \leq j. V_{r_k} = S_{th}(m_p(r_k)),$	12
$\forall q_k \in m_p^{-1}(P). V_{q_k} \cup V_{p_k} \leq \text{upperbound}(q_k),$	13
$\forall r_k \in m_p^{-1}(P'). V_{r_k} + 1 \leq \text{upperbound}(r_k)$	14
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{t(n)} \langle S_n[n \mapsto \text{idle}], S_a[b \mapsto \text{idle}], S_{th}[p_1 \mapsto \emptyset] \dots [p_i \mapsto \emptyset]$	15
$[m_p^{-1}(q_1) \mapsto V_{q_1} \cup V_{p_1}] \dots [m_p^{-1}(p_i) \mapsto V_{p_i} \cup V_{p_i}]$	16
$[m_p^{-1}(r_1) \mapsto V_{r_1} \cup \{\text{null}\}] \dots [m_p^{-1}(r_j) \mapsto V_{r_j} \cup \{\text{null}\}], S_\Sigma \rangle$	17

Token consumption by streaming parameters during activity execution: This rule passes tokens to the corresponding activity that are offered to the *CallBehaviorAction*. Only tokens offered in streaming parameters and belonging to the P_s are consumed in this rule (Lines 4,6). APN with incoming tokens that have not been set before are removed from the set P_n , since a empty P_n is required for the termination of the activity (Line 13).

$a \in A, S_a(a) = \text{idle},$	1
$P_{ocu} = \{th \in APN(a) S_{th}(th) \neq \emptyset\},$	2
$P_s \in PS(a). \forall p \in P_{ocu}. p \in P_s \wedge P_{ocu} \neq \emptyset,$	3
$\{n_1, \dots, n_i\} = \{n \in \text{node}(a) \text{type}(n) = \text{InitialNode}\},$	4
$\{m_1, \dots, m_j\} = \{m \in \text{node}(a) \text{type}(m) = \text{AcceptEventAction} \wedge \text{inpin}(m) = \emptyset\}$	5
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \xrightarrow{i(n)} \langle S_n[m_1 \mapsto \langle \text{executing}, \emptyset \rangle] \dots [m_j \mapsto \langle \text{executing}, \emptyset \rangle],$	6
$S_a[a \mapsto \langle \text{executing}, P_s, P_s - P_{ocu} \rangle],$	7
$S_{th}[\text{outpin}(n_1) \mapsto \{CT\}] \dots [\text{outpin}(n_j) \mapsto \{CT\}], S_\Sigma \rangle$	8

OutParameter: This rule transfers tokens that have been offered to *APN*, which are necessary to the termination of the *Activity*.

$a \in A. S_a(a) = \langle \text{executing}, P_s, P_n \rangle,$	1
$e \in \text{edge}(a). e = \langle s, t, g \rangle \wedge t \in APN(a) \wedge s \in \text{pin}(a) \wedge \text{isExeption}(t) = \text{FALSE} \wedge$	2
$\text{transfer}(e) \neq \emptyset,$	3
$Vc = \text{transfer}(e),$	4
$Vs = \{V S_{th}(\text{source}(e)) = V\},$	5
$Vt = \{V S_{th}(\text{target}(e)) = V\}$	6
<hr/>	
$\langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n, S_a, S_{th}[\text{source}(e) \mapsto Vs] Vc]$	7
$[\text{target}(e) \mapsto Vt \cup Vc], S_\Sigma \rangle$	8

5.14 Exceptions

Throwing an exception: An *Activity* is in an exception if an exception parameter has received a token.

$$\begin{array}{l}
 a \in A, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
 e \in \text{edge}(a).e = \langle s, t, g \rangle \wedge t \in \text{APN}(a) \wedge s \in \text{pin}(a) \wedge \text{isException}(t) = \text{TRUE} \wedge \\
 \text{transfer}(e) \neq \emptyset, \\
 V_e = \text{transfer}(e), \\
 \{p_1, \dots, p_i\} = \text{pin}(a), \\
 \{n_1, \dots, n_j\} = \text{node}(a) \\
 \hline
 \langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n[n_1 \mapsto \text{idle}] \dots [n_j \mapsto \text{idle}], S_a[a \mapsto \text{exception}(V_e)], \\
 S_{th}[p_1 \mapsto \emptyset] \dots [p_i \mapsto \emptyset], S_\Sigma \rangle
 \end{array}
 \begin{array}{r}
 1 \\
 2 \\
 3 \\
 4 \\
 5 \\
 6 \\
 7 \\
 8
 \end{array}$$

Inkovation of the handler of an exception: After the exception is thrown, the exception handler is invoked if there is any.

$$\begin{array}{l}
 a, b \in A, n \in \text{node}(a), S_n(n) = \langle \text{idle}, \emptyset \rangle, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
 S_a(b) = \langle \text{exception}, v_e \rangle, \text{isSynchronous}(n) = \text{true}, S_n(n) = \langle \text{idle}, \emptyset \rangle, \\
 n \in \text{handler}(b).\text{typeException}(n) = \delta(v_e), \\
 \forall q \in \text{outpin}(h).S_{th} = \emptyset \\
 p = \text{inpin}(h) \\
 \hline
 \langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n, S_a, S_{th}[p \mapsto v_e], S_\Sigma \rangle
 \end{array}
 \begin{array}{r}
 1 \\
 2 \\
 3 \\
 4 \\
 5 \\
 6
 \end{array}$$

Propagation of the exception in case of no handler: In case that there is no exception handler, the exception is propagate to the next hierarchical level.

$$\begin{array}{l}
 a, b \in A, n \in \text{node}(a), \text{type}(n) = \text{CallBehaviorAction}, b = \text{behavior}(n), \\
 S_a(b) = \langle \text{exception}, v_e \rangle, S_n(n) = \langle \text{executing}, \emptyset \rangle, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
 \text{isSynchronous}(n) = \text{true}, \\
 \nexists n \in \text{handler}(b).\text{typeException}(n) = \delta(v_e), \\
 \{p_1, \dots, p_i\} = \text{pin}(a), \\
 \{n_1, \dots, n_j\} = \text{node}(a) \\
 \hline
 \langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n[n_1 \mapsto \text{idle}] \dots [n_j \mapsto \text{idle}], S_a[a \mapsto \text{exception}(V_e)][b \mapsto \text{idle}], \\
 S_{th}[p_1 \mapsto \emptyset] \dots [p_i \mapsto \emptyset], S_\Sigma \rangle
 \end{array}
 \begin{array}{r}
 1 \\
 2 \\
 3 \\
 4 \\
 5 \\
 6 \\
 7 \\
 8 \\
 9
 \end{array}$$

Termination of the exception due to the asynchronous call: If the *Activity* has been asynchronously invoked and the exception has not been hadler, the exception is deleted.

$$\begin{array}{l}
 a, b \in A, n \in \text{node}(a), S_n(n) = \langle \text{idle}, \emptyset \rangle, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \\
 S_a(b) = \langle \text{exception}, v_e \rangle, \text{isSynchronous}(n) = \text{false} \\
 \hline
 \langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n, S_a[b \mapsto \text{idle}], S_{th}, S_\Sigma \rangle
 \end{array}
 \begin{array}{r}
 1 \\
 2 \\
 3 \\
 4
 \end{array}$$

Transferring outgoing tokens from exception handler: After a handler terminates the execution, its results are tranfered to the *Activity* that threwed the exception.

$$\begin{array}{l}
a, b \in A, n, m \in \text{node}(a), S_n(n) = \langle \text{idle}, \emptyset \rangle, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, S_a(b) = \\
\langle \text{exception}, v_e \rangle, \\
\text{isSynchronous}(n) = \text{true}, \text{type}(m) = \text{CallBehaviorAction}, b = \text{behavior}(m), \\
n \in \text{handler}(b). \text{typeException}(n) = \delta(v_e), \\
\exists p \in \text{outpin}(n). S_{th} \neq \emptyset \\
\{p_1, \dots, p_i\} = \text{outpin}(a), \\
\forall 1 \leq k \leq i. q_k = m_e(p_k), \\
\forall 1 \leq k \leq i. V_{p_k} = S_{th}(p_k), \\
\forall 1 \leq k \leq i. V_{q_k} = S_{th}(q_k), \\
\{r_1, \dots, r_j\} = \{r \in \text{outpin}(m) \mid \nexists q_k, q_k = r\}, \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightarrow \langle S_n, S_a[b \mapsto \text{idle}], S_{th}[p_1 \mapsto \emptyset] \dots [p_i \mapsto \emptyset][q_1 \mapsto V_{q_1} \cup V_{p_1}] \dots [q_i \mapsto V_{p_i} \cup V_{q_i}] \\
[r_1 \mapsto \{CT\}] \dots [r_j \mapsto \{CT\}], S_\Sigma \rangle
\end{array}
\begin{array}{r}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
8 \\
9
\end{array}$$

6 Semantics extensions

Semantics extensions refer to the extension of the core semantics in order to define a specific interpretation of UML activities. Depending on the extension, possible behaviors given by the core semantics may be reduced or enlarged, which can affect the consistency with the standard. This section shows some examples that extend the semantics described in previous section. Extensions in the execution time and single core processing are based on the profile DMOSES [8], which adds information to UML models regarding execution time, parallelism and priority in order to generate software for embedded systems. Previous work [4] presented a approach to determine if an extension to the semantics is consistent with the standard that is based on the simulation ordering of Park.

6.1 Execution time

For real-time systems, violations of time deadlines can lead to significant errors in the system. Therefore, information about the duration of the execution of nodes has to be modeled in order to be able to verify timing constraints. Since the UML standard does not specify the duration of the execution of nodes, timing information can be added to the model using profiling. The following inference rule $\text{exeTime}(n)$ defines the time that has elapsed since the invocation of the action $(i(n))$, where the clock is started. The semantics of the clocks process implements the time increasing and ensures a correct termination. This rule adds a new label to the semantics that does not exist in the reference semantics. Note that a set of clocks is added to the state of the system and the status *ready* is added to the node. The conclusion of the rule stops the timer corresponding to the node that enables a termination rule.

$$\begin{array}{lcl}
a \in A, n \in \text{node}(a), \text{type}(n) = \text{Action}, S_a(a) = \langle \text{executing}, P_s, P_n \rangle, & & 1 \\
S_n(n) = \langle \text{executing}, f_{in} \rangle, & & 2 \\
C(n) \geq \text{executionTime}(n) & & 3 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma, C \rangle \xrightarrow{\text{exeTime}(n)} \langle S_n[n \mapsto \langle \text{ready}, f_{in} \rangle], S_a, S_{th}, S_\Sigma, C[n \mapsto \perp] \rangle & & 4
\end{array}$$

6.2 Non-preemptive single core

Activities implemented by non-preemptive single core processors are a good example of how a behavior can be limited by the target platform. Although multiple nodes can be being executed at the same time in a UML activity, a single core processor can only execute one at the time. The following premises are added to all invocation rules ($i(n)$) to specify non-preemptive single core processing behavior. These premises limit the applicability of the invocation rules since no node can be invoked if there is another being executed.

$$\forall a \in A. \nexists m \in \text{node}(a). m \neq n \wedge S_n(m) = \langle \text{executing}, f'_{in} \rangle$$

6.3 Implementations of token consumption

The UML standard defines that tokens can be transferred only if they can be immediately consumed. This behavior can be difficult or impossible to implement depending on the target domain. Variations in the token consumption can lead to drastic changes in the entire behavior of the activity. Therefore, it is important to include these variations in the semantics in order to accurately verify the system. This paper analyzes two variations that perform token transfer separately from token consumption (Figure 3). Variation 1 sequentially transfers all possible tokens from output pins to input pins before any node execution. Variation 2 performs both token transfer and node execution (if available) in a non-deterministic way.

In order to formally specify these variations, a *micro-step* has to be added to the semantics, which is responsible for the token transfer, as is shown in the following rule. Furthermore, the token consumption of the *macro-steps* of the reference semantics has to be limited to the input pins.

$$\begin{array}{lcl}
a \in A, e \in \text{edge}(a), S_a(a) = \langle \text{executing}, P_s, P_n \rangle, \text{transfer}(e) \neq \emptyset, & & 1 \\
V_c = \text{transfer}(e), & & 2 \\
V_s = S_{th}(\text{source}(e)), & & 3 \\
V_t = S_{th}(\text{target}(e)) & & 4 \\
\hline
\langle S_n, S_a, S_{th}, S_\Sigma \rangle \rightsquigarrow \langle S_n, S_a, S_{th}[\text{source}(e) \mapsto V_s \upharpoonright V_c][\text{target}(e) \mapsto V_t \cup V_c], S_\Sigma \rangle & & 5
\end{array}$$

Additionally for Variation 1, line 1 of the definition of a *transition* has to be changed in order to ensure that all token transfers are performed before any execution takes place. The extended line is shown below.

$$\langle S_n, S_a, S_{th}, S_\Sigma \rangle (\rightsquigarrow)^* \langle S'_n, S'_a, S'_{th}, S'_\Sigma \rangle \not\rightsquigarrow \langle S'_n, S'_a, S'_{th}, S'_\Sigma \rangle \rightarrow \langle S''_n, S''_a, S''_{th}, S''_\Sigma \rangle$$

A part of the state-space of the activity A with different token consumption semantics is shown in figure 3. Variation 1 presents the most limited behavior, followed by the reference semantics. Note that the action C is never executed in Variation 1 (denoted by $i(C)$). The reference semantics can finalize in two states, in which either C or D has been executed since both are competing for an outgoing token of A . By contrast, Variation 2 has three final states. In two of them, D is executed, one where the token between B and C has been transferred and one where not. Variation 2 enables more behaviors, since the position of the tokens is indirectly taken into account.

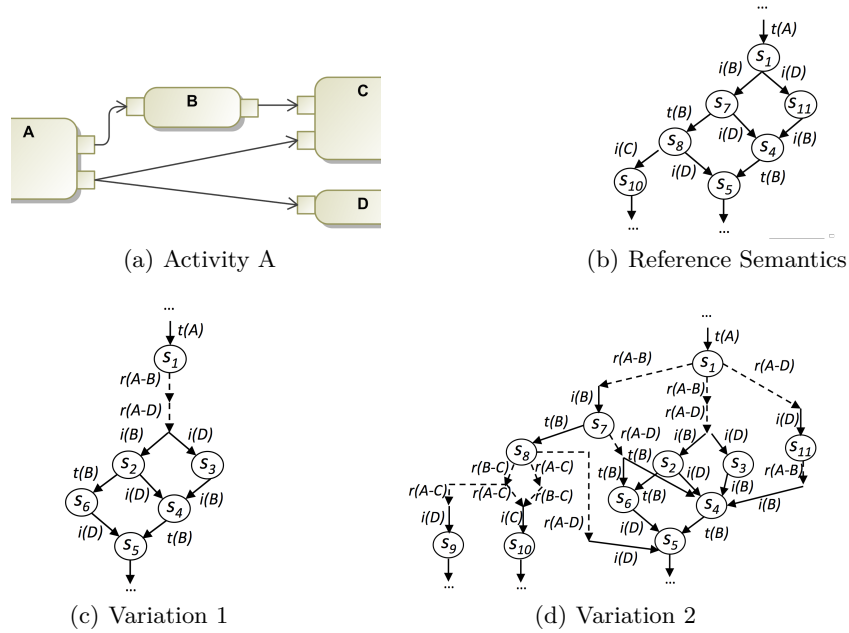


Fig. 3. The implementation of token consumption can reduce or increase the number of possible behaviors

7 Conclusion

This paper has presented a reference operational semantics for UML 2.x activity diagrams. The presented semantics aims to provide the same flexibility and

extensibility as the standard, and is defined by inference rules that specify all allowed behaviors by the standard. These rules can be extended or additional rules can be added in order to customize the semantics to accurately specify characteristics of the system implementation and the domain requirements. Since the semantics is operational, it also serves as a basis for translating activity diagrams into model-checking tools. The correctness of the semantics cannot be formally verified, since the standard does not provide a formal reference and there is not yet a consensus formal semantics for UML activity diagrams. Therefore, we also want with our extensible semantics to open a discussion in the community about a common formal framework for UML models.

For future work, we want to implement the generation of Kripke structures based on the presented inference rules, thereby facilitating model checking of activities by translating the resulting structures to model-checker languages. In addition, although the proposed semantics can specify parallelism between executions of multiple nodes, the start and termination of multiple executions are interleaved. In order to support full parallelism, we are studying methods to extend the step semantics using notions from partial-order reduction. These methods can be applied to the resulting Kripke structure, which has information about the dependency between elements caused by shared resources.

References

- [1] Börger, E., Cavarra, A., Riccobene, E.: An ASM Semantics for UML Activity Diagrams, vol. 1816, pp. 293–308. Springer Berlin Heidelberg (2000)
- [2] Broy, M., Cengarle, M.: Uml formal semantics: lessons learned. *Software & Systems Modeling* 10(4), 441–446 (2011)
- [3] Cao, H., Ying, S., Du, D.: Towards model-based verification of bpm with model checking. In: *Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*. pp. 190–190 (Sept 2006)
- [4] Daw, Z., Cleaveland, R.: An extensible operational semantics for uml activity diagrams. In: *International Conference on Software Engineering and Formal Methods* (2015)
- [5] Daw, Z., Cleaveland, R.: A semantics-based framework for uml simulation and verification. In: *Model Driven Engineering Languages and Systems* (2016 - submitted)
- [6] Daw, Z., Cleaveland, R., Vetter, M.: Integrating of model checking and uml based model-driven development for embedded systems. In: *Automated Verification of Critical Systems (AVOCS), Electronic Communications of the EASS* (2013)
- [7] Daw, Z., Cleaveland, R., Vetter, M.: Formal verification of software-based medical devices considering medical guidelines. *International Journal of Computer Assisted Radiology and Surgery* 9(1), 145–153 (2014)
- [8] Daw, Z., Vetter, M.: Deterministic UML Models for Interconnected Activities and State Machines, vol. 5795, pp. 556–570. Springer Berlin Heidelberg (2009)
- [9] Eshuis, R.: Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.* 15(1), 1–38 (2006)
- [10] Grobelna, I., Grobelny, M., Adamski, M.: Petri nets and activity diagrams in logic controller specification - transformation and verification. In: *Mixed Design*

- of Integrated Circuits and Systems (MIXDES), 2010 Proceedings of the 17th International Conference. pp. 607–612 (June 2010)
- [11] Grönniger, H., Reiß, D., Rumpe, B.: Towards a semantics of activity diagrams with semantic variation points. In: MODELS. pp. 331–345. Springer-Verlag, Berlin, Heidelberg (2010)
 - [12] Guelfi, N., Mammar, A.: A formal semantics of timed activity diagrams and its promela translation. In: Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific. pp. 8 pp.– (Dec 2005)
 - [13] Gulan, S., Johr, S., Kretschmer, R., Rieger, S., Ditze, M.: Graphical modelling meets formal methods. In: Industrial Informatics (INDIN), 2013 11th IEEE International Conference on. pp. 716–721 (July 2013)
 - [14] Knieke, C., Schindler, B., Goltz, U., Rausch, A.: Defining domain specific operational semantics for activity diagrams. Tech. rep., TU Clausthal (2012)
 - [15] Lano, K.: UML 2 semantics and applications. Wiley Online Library (2009)
 - [16] OMG: Unified Modeling Language, Superstructure, Version 2.4.1. <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF> (2011)
 - [17] Staines, T.: Intuitive mapping of uml 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In: 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. pp. 191–200 (March 2008)
 - [18] Störrle, H.: Structured nodes in uml 2.0 activities. Nordic Journal of Computing 11(3), 279–302 (2004)
 - [19] Störrle, H.: Semantics and verification of data flow in uml 2.0 activities. Electronic Notes in Theoretical Computer Science 127(4), 35–52 (2005)
 - [20] Störrle, H., Hausmann, J.: Towards a formal semantics of uml 2.0 activities. In: In Proceedings German Software Engineering Conference. vol. 64 (2005)
 - [21] Vitus S. W, L.: A formalism for reasoning about uml activity diagrams. In: Nordic Journal of Computing. vol. 14 (January 2007)
 - [22] Xu, D., Liu, Z., Liu, W.: Towards formalizing uml activity diagrams in csp. In: International Symposium on Computer Science and Computational Technology. vol. 2 (2008)